Exception: a transfer of control to the OS kernel in response to some event

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to **"next" instruction**

Timer interrupt: take control back from user programs to kernel eg. context switch

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - Intentional
    - Examples: system calls (requests for services from the kernel)
    - Returns control to **"next" instruction**
  - **Faults**
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable)
    - Either re-executes faulting **("current") instruction or aborts**
  - **Aborts**
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error (data error/inconsistency detected), machine check (hardware issue detected)
    - **Aborts current program**

System Calls

Request a service that is not accessible for program from OS

- Each x86-64 system call has a unique ID number (assigned by the operating system)
- Examples:

Number	Name	Description
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

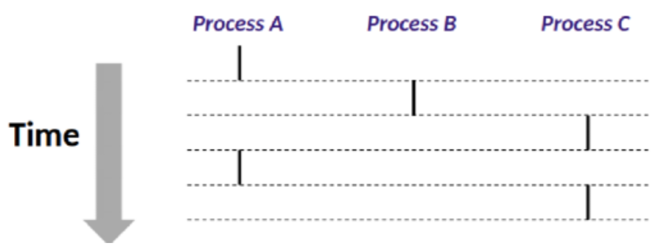
Process: an instance of a running program

Abstraction (Illusion):

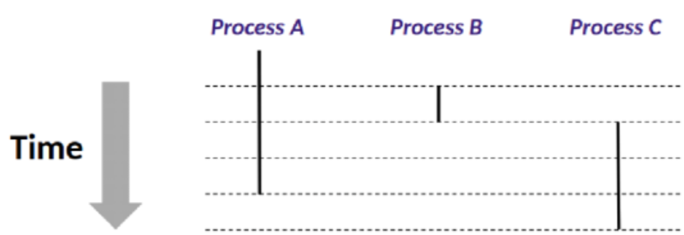
- **Logical control flow**
  - Each program seems to have exclusive use of the CPU
  - Provided by kernel mechanism called context switching
- **Private address space**
  - Each program seems to have exclusive use of main memory.
  - Provided by kernel mechanism called virtual memory

Concurrent Processes

- Each process is a logical control flow.
- Two processes run **concurrently** (are concurrent) if their flows overlap in time
- Otherwise, they are **sequential**
- Examples (running on a single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C

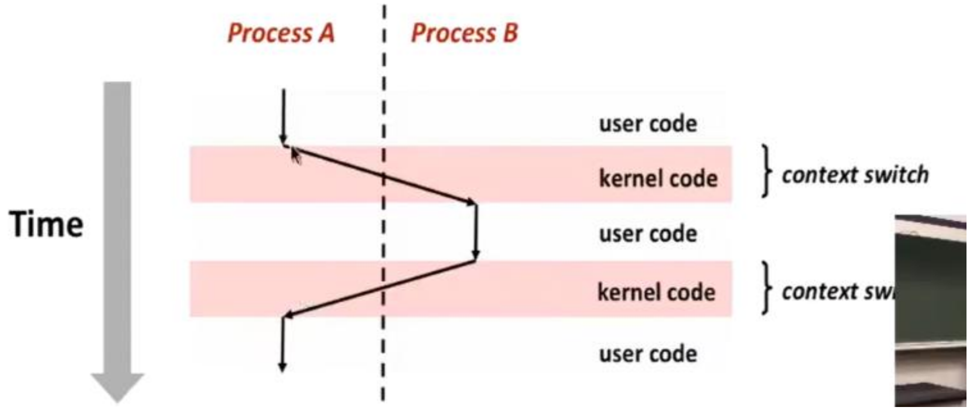


View of Concurrent Processes

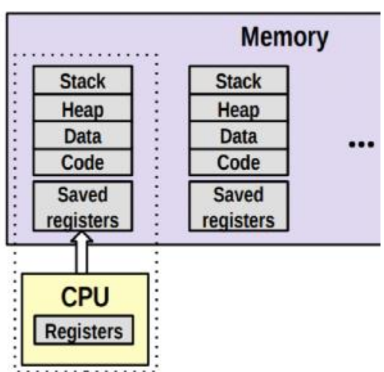


Context Switching

- **Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **Control flow passes from one process to another via a *context switch***

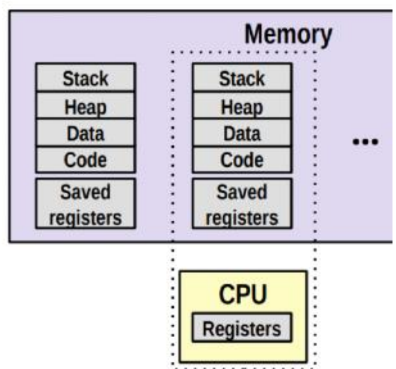


1



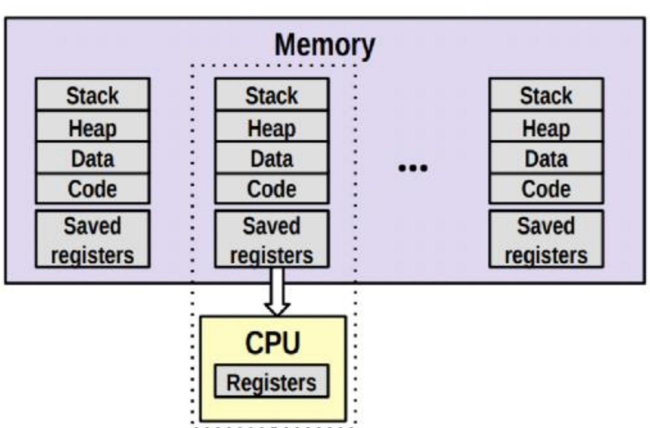
■ Save current registers in memory

2



■ Schedule next process for execution

3



■ Load saved registers and switch address space (**context switch**

1. void exit (int status) - terminate the process, called once but never returns

2.
- **Parent process creates a new running child process by calling `fork`**
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is **almost** identical to parent:
    - Child gets an identical (but separate) copy of the parent's virtual address space (this includes all the data on the stack and on the heap, and all the instructions) .
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent
- `fork(...)` function is interesting (and often confusing) because it is called once but returns twice

3.
Reaping Child Processes

- Idea
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a "**zombie**"
    - Living corpse, half alive and half dead
- **Reaping** (harvesting, collecting)
  - Performed by parent on terminated child process (using `wait` or `waitpid`)
  - Parent is given exit status information (it is notified that the child process terminated and, by receiving the exit status, it acknowledges the termination)
  - Kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child process will be reaped by init process (pid == 1) (*root of the process three*)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers
  - (although you should be a good citizen and collect your zombies if possible)

`wait`: Synchronizing with Children

- Parent reaps a child by calling the wait function
- `int wait(int *child_status)`
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated

only wait its own child (not grandchild)

Parent instructions after wait will processed when child terminates

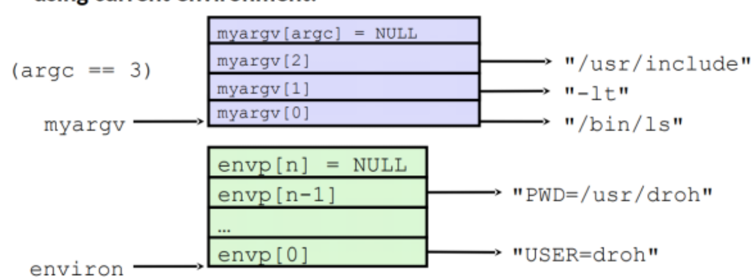
4.

execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process:
  - Executable file filename
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - ...with argument list **argv**
    - By convention `argv[0]==filename`
  - ...and environment variable list envp
    - "name=value" strings (e.g., USER=droh)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
  - (the current process is gone, it is now running a different program)
- Called once and never returns
  - ...except if there is an error

execve Example

- **Executes** `"/bin/ls -lt /usr/include"` **in child process using current environment:**




```
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
```

5. Kill(pid, SIGUSR1); kill(pid, SIGKILL); kill(pid, SIGSTOP)
   Send signal to pid